



# Buenas prácticas de programación en Python

**Lección 1: Control de errores, pruebas y validación de datos.**

# ÍNDICE

<b>Control de errores, pruebas y validación de datos.....</b>	<b>2</b>
<b>Presentación y objetivos.....</b>	<b>2</b>
<b>1. Control de errores.....</b>	<b>3</b>
1.1 Errores de sintaxis .....	3
1.2 Errores lógicos.....	5
1.3 Excepciones .....	6
1.4 Manejo de excepciones .....	8
1.5 Excepciones definidas por el usuario.....	12
<b>2. Pruebas y validación de datos.....</b>	<b>14</b>
2.1 Técnicas de programación defensiva.....	15
2.2 Validaciones: caso práctico .....	17
2.3 Comprobaciones por contenido .....	17
2.4 Comprobación de valores de entrada del usuario .....	18
2.5 Comprobaciones por tipo .....	19
2.6 Comprobaciones por tamaño de cadena.....	20
<b>3. Puntos clave.....</b>	<b>21</b>

# Control de errores, pruebas y validación de datos.

## PRESENTACIÓN Y OBJETIVOS

En esta lección aprenderemos a crear códigos de programación en los que utilizaremos controles de errores y pruebas para desarrollar códigos más eficientes e interpretables. Por otro lado, estudiaremos la importancia de realizar pruebas y validación en los datos que son utilizados como entradas de los programas que implementemos. Validar y verificar que nuestro programa es capaz de responder de una forma favorable ante cualquier tipo de entrada aumenta la calidad de nuestro software.



### **Objetivos**

- Conocer la importancia del control de errores
- Saber gestionar y depurar los errores de un código de programación.
- Conocer el concepto de programación defensiva y cómo aplicarlo en nuestro día a día.
- Conocer los diferentes mecanismos y herramientas disponibles para validar la entrada de datos en nuestro código Python.

## 1. CONTROL DE ERRORES

Hasta ahora, no le hemos dedicado mayor importancia a los mensajes de error que hemos obteniendo a la hora de realizar nuestros programas. Hemos utilizado estos mensajes de error para interpretar y tratar de entender qué estaba ocurriendo con nuestro código, por qué no funcionaba e intentar solucionar el error.

Podemos distinguir entre dos tipos de errores:

- Errores de sintaxis
- Errores lógicos

### 1.1 Errores de sintaxis

Este tipo de errores son quizá los más conocidos que nos encontramos cuando estamos aprendiendo a programar en Python, este tipo de errores muestran un mensaje donde indican que o bien no se ha escrito correctamente un comando de Python o que por algún motivo nos estamos “saltando” las reglas de Python. Por ejemplo, en la siguiente captura se muestra un ejemplo de error de sintaxis, en el que el alumno podrá comprobar que el error surge por no haber escrito los dos puntos (':') después de la orden `if`.

El analizador sintáctico de Python repite la línea de programación donde ha encontrado el error y muestra una pequeña flecha que apunta al último punto de la línea en la que se encontró el error. En este caso, el analizador sintáctico nos indica que el error ha sido detectado en la función `print()`, ya que hemos olvidado escribir los dos puntos (':') después del comando `if`. Nótese que el analizador sintáctico también muestra la línea donde se ha producido el error, con el objetivo de facilitar la tarea al programador a la hora de solucionar el problema.

```
ramon@ramon-rd: ~/Escritorio/ev
Python 3.8.5 (default, Sep  4 2020, 07:30:14)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> if(5 > 2) print("El primer operador es mayor")
      File "<stdin>", line 1
        if(5 > 2) print("El primer operador es mayor")
            ^
SyntaxError: invalid syntax
>>>
```

Siguiendo esta línea, a continuación se muestra una captura con un error de sintaxis en un código de programación. ¿Podrías adivinar a qué se debe el error?

```
ramon@ramon-rd: ~/Escritorio/ev
>>> a = 5
>>> b = 3
>>> print(g'{a} es mayor que {b}')
      File "<stdin>", line 1
        print(g'{a} es mayor que {b}')
            ^
SyntaxError: invalid syntax
>>>
```

En este caso, el origen del error proviene dentro de la orden `if`, en concreto de la letra `g` que hemos escrito antes del mensaje a mostrar. La sintaxis correcta sería escribir una `f`, tal y como se muestra en la siguiente captura:



```
ramon@ramon-rd: ~/Escritorio/ev
>>> a = 5
>>> b = 3
>>> print(f'{a} es mayor que {b}')
5 es mayor que 3
>>>
```

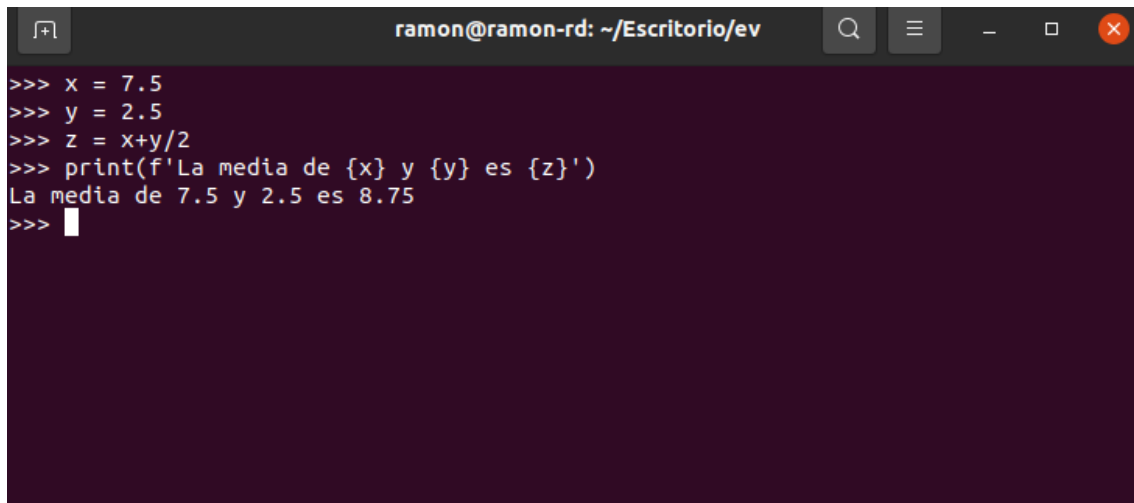


### ***Presta atención***

Debemos aprovechar las herramientas que nos brinda el analizador sintáctico de Python, pues nos ayuda a depurar y corregir nuestro código de una forma más sencilla.

## **1.2 Errores lógicos**

Este tipo de errores son los más difíciles de detectar ya que darán resultados impredecibles o incluso podría llegar a bloquear el programa. No obstante, este tipo de errores son muy fáciles de solucionar ya que podemos hacer uso de un depurador que ayudará a solucionar este tipo de problemas. A continuación se muestra un ejemplo de un error de tipo lógico:



```
ramon@ramon-rd: ~/Escritorio/ev
>>> x = 7.5
>>> y = 2.5
>>> z = x+y/2
>>> print(f'La media de {x} y {y} es {z}')
La media de 7.5 y 2.5 es 8.75
>>>
```

Como se puede comprobar, el programa no devuelve ningún error. Sin embargo, es obvio que la media de los números 7.5 y 2.5 es 5, y no 8.75 como muestra el programa del ejemplo anterior.

En estos casos, decimos que el error es de tipo lógico ya que no hay ningún problema con la propia sintaxis de Python y el programa funciona (entendiendo funcionar como que el intérprete ha sido capaz de traducir el programa a código máquina y mostrar un resultado, aunque en este caso el resultado no es el correcto).

Este tipo de errores aprenderemos a resolverlos más adelante, haciendo uso del depurador de Python.

### 1.3 Excepciones

Como hemos visto hasta ahora, incluso si un programa es sintácticamente correcto, puede producirse un error cuando se intenta ejecutar. Los errores detectados durante la ejecución se denominan excepciones. La mayoría de las excepciones no son manejadas por los programas y dan como resultado mensajes de error como los siguientes:

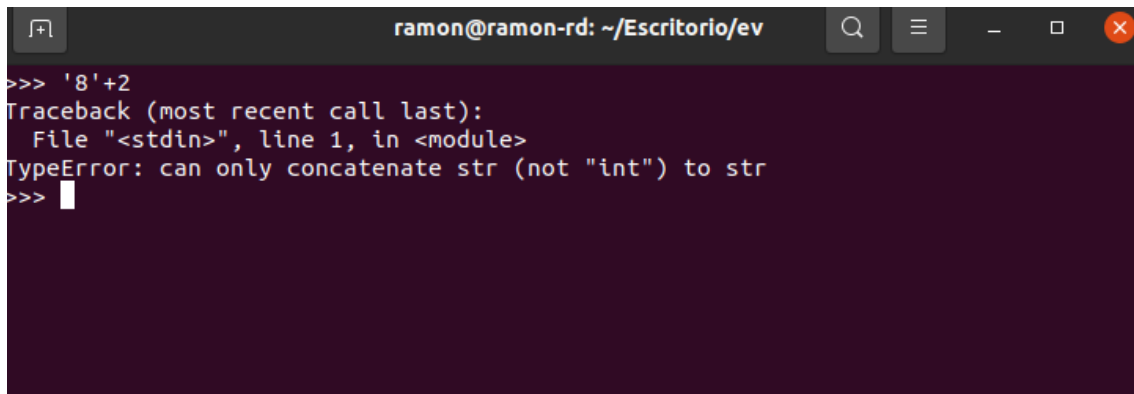
```
ramon@ramon-rd: ~/Escritorio/ev
>>> 2/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 
```

En este caso, la excepción "**ZeroDivisionError**" hace referencia a un error relacionado por dividir un número por el valor 0. Como bien es sabido, la operación de dividir entre 0 no está permitida.

```
ramon@ramon-rd: ~/Escritorio/ev
>>> 7+numero*5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'numero' is not defined
>>> 
```

En este caso, la excepción "**NameError**" viene dada porque estamos haciendo uso de una variable que no ha sido declarada previamente, y por tanto no es conocida en la ejecución del programa.



A screenshot of a terminal window with a dark background. The window title is 'ramon@ramon-rd: ~/Escritorio/ev'. The prompt is '>>>'. The user has entered the expression ''8'+2'. The terminal shows a traceback: 'Traceback (most recent call last):', 'File "<stdin>", line 1, in <module>', and 'TypeError: can only concatenate str (not "int") to str'. The prompt '>>>' is followed by a cursor.

```
>>> '8'+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
>>> █
```

Por último, la excepción **"TypeError"** viene dada por hacer operaciones no permitidas entre diferentes tipos de datos. En el ejemplo anterior el error puede ser arreglado de dos formas:

- Escribir 2 entre comillas ('2') de modo que el operador + se utiliza como operador de concatenación, y el resultado sería concatenar dos caracteres, obteniendo como resultado '82'
- Eliminar las comillas simples del número 8. En este caso el operador + sería utilizado como el operador matemático suma, obteniendo como resultado el valor 10.

Como se puede comprobar en las excepciones anteriores, el inicio del mensaje muestra el tipo de excepción invocado y a la derecha de la excepción se muestra el mensaje de error que muestra el contexto en el que surge la excepción.

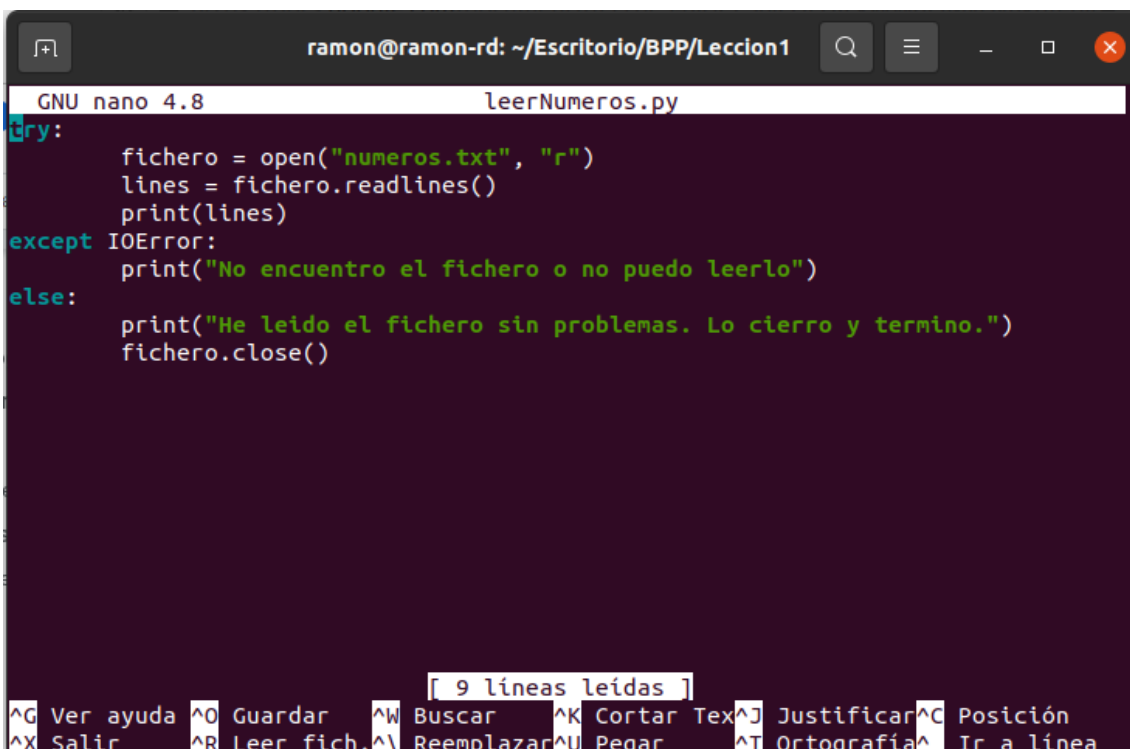
## 1.4 Manejo de excepciones

Es posible desarrollar un programa capaz de manejar un conjunto de excepciones. Las excepciones en Python pueden ser gestionadas por medio de las órdenes "try" y "except". Si está desarrollando un código en el que cree que podrían surgir excepciones, puede "proteger" el programa incluyendo las órdenes "try" y "except". De este modo, si ocurre una excepción en su programa, en lugar de detenerse mostrando un error, le mostrará un mensaje de que ha ocurrido la excepción pero tiene la posibilidad de continuar con la ejecución del programa hasta el final.

La sintaxis es la siguiente:

```
try:
    operaciones deseadas
    ...
except ExceptionI:
    Si ocurre la primera excepción, ejecuta este bloque
except ExceptionII:
    Si ocurre la segunda excepción, ejecuta este bloque
    ....
else:
    Si no hay ninguna excepción, haz esto otro.
```

A continuación se muestra un programa que se encarga de leer un fichero con un conjunto de números y los muestra por pantalla. Si el fichero no existe o no puede ser abierto, mostraremos un mensaje personalizado que indique dicho error por pantalla:

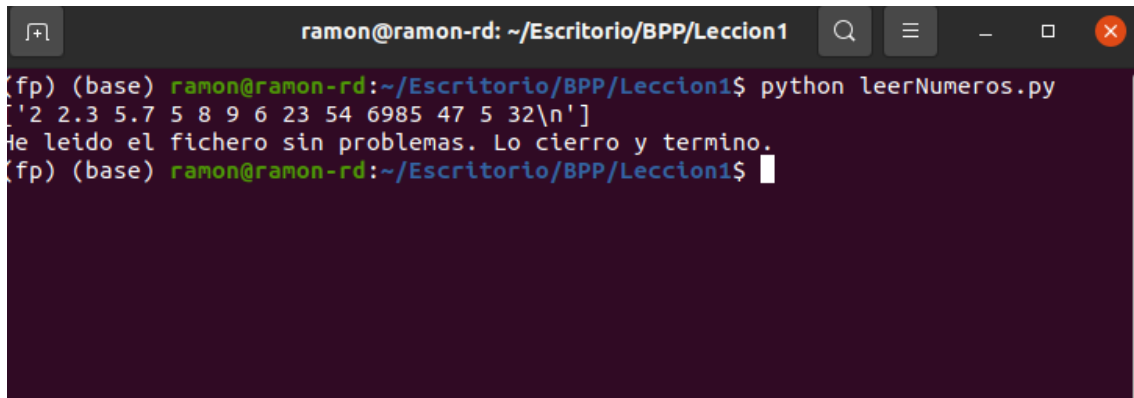


The screenshot shows a terminal window titled "ramon@ramon-rd: ~/Escritorio/BPP/Leccion1". The window contains the GNU nano 4.8 editor editing a file named "leerNumeros.py". The code in the file is as follows:

```
try:
    fichero = open("numeros.txt", "r")
    lines = fichero.readlines()
    print(lines)
except IOError:
    print("No encuentro el fichero o no puedo leerlo")
else:
    print("He leído el fichero sin problemas. Lo cierro y termino.")
    fichero.close()
```

At the bottom of the terminal, a status bar indicates "[ 9 líneas leídas ]" and a menu of keyboard shortcuts is visible, including Ver ayuda, Guardar, Buscar, Cortar Text, Justificar, Posición, Salir, Leer fich., Reemplazar, Pegar, Ortografía, and Ir a línea.

En el ejemplo anterior, leemos el fichero con el nombre “numeros.txt” y lo mostramos por pantalla. Sin conseguimos hacerlo sin problemas, mostramos un mensaje en el que indicamos que el proceso ha ido bien.

A terminal window titled 'ramon@ramon-rd: ~/Escritorio/BPP/Leccion1' with search, menu, and window control icons. The prompt is '(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1\$'. The command 'python leerNumeros.py' has been executed. The output is: '[2 2.3 5.7 5 8 9 6 23 54 6985 47 5 32\n]' followed by the message 'He leído el fichero sin problemas. Lo cierro y termino.' and the prompt '(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1\$' with a cursor.

```
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$ python leerNumeros.py
[2 2.3 5.7 5 8 9 6 23 54 6985 47 5 32\n]
He leído el fichero sin problemas. Lo cierro y termino.
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$
```

En caso contrario, si el programa no pudiera encontrar o abrir el fichero indicado, mostrará un mensaje personalizado en el que indicamos que no hemos podido leer el fichero.

A terminal window titled 'ramon@ramon-rd: ~/Escritorio/BPP/Leccion1' with search, menu, and window control icons. The prompt is '(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1\$'. The command 'python leerNumeros.py' has been executed. The output is the message 'No encuentro el fichero o no puedo leerlo' and the prompt '(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1\$' with a cursor.

```
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$ python leerNumeros.py
No encuentro el fichero o no puedo leerlo
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$
```

Si lo deseamos, también podemos mostrar qué tipo de excepción a ocurrido junto con el mensaje personalizado que hemos escrito, para ser más precisos ayudar a terceros a entender mejor nuestro código:

```

ramon@ramon-rd: ~/Escritorio/BPP/Leccion1
GNU nano 4.8 leerNumeros.py
try:
    fichero = open("numeros.txt", "r")
    lines = fichero.readlines()
    print(lines)
except IOError as err:
    print("No encuentro el fichero o no puedo leerlo. Error: ", err)
else:
    print("He leído el fichero sin problemas. Lo cierro y termino.")
    fichero.close()

[ 9 líneas leídas ]
^G Ver ayuda ^O Guardar ^W Buscar ^K Cortar Tex ^J Justificar ^C Posición
^X Salir ^R Leer fich. ^\ Reemplazar ^U Pegar ^T Ortografía ^_ Ir a línea
  
```

De este modo, el mensaje de error que obtenemos si se activa la excepción es el siguiente:

```

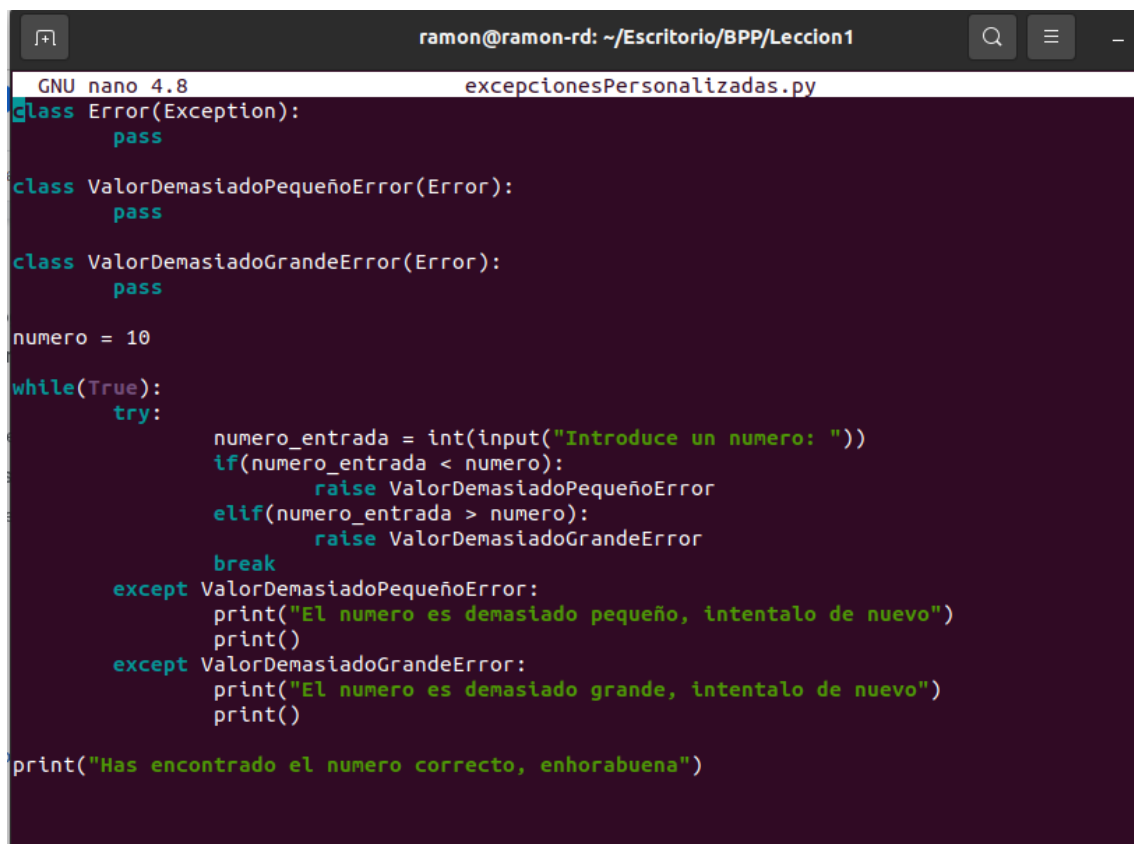
ramon@ramon-rd: ~/Escritorio/BPP/Leccion1
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$ python leerNumeros.py
No encuentro el fichero o no puedo leerlo. Error: [Errno 2] No such file or dir
ectory: 'numeros.txt'
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$
  
```

Indicándonos que el programa no ha sido capaz de encontrar el fichero de datos a leer.

## 1.5 Excepciones definidas por el usuario

Una opción muy interesante que ofrece el lenguaje de programación Python es la posibilidad de crear excepciones personalizadas. Las excepciones que podemos crear derivan de la clase *Exception* directa o indirectamente.

Este tipo de clases se pueden implementar como cualquier otro tipo de clases, pero en la práctica se tratan de desarrollar de la forma más simple y directa posible. Muchas implementaciones declaran una clase principal y el resto de clases derivan de la clase base o principal. Para clarificar este concepto, se muestra un ejemplo sencillo en el que creamos una excepción personalizada, en la que trataremos de crear unas excepciones que sean lanzadas cuando un número ingresado por el usuario sea mayor o menor que un umbral determinado.



```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion1
GNU nano 4.8 excepcionesPersonalizadas.py
class Error(Exception):
    pass

class ValorDemasiadoPequeñoError(Error):
    pass

class ValorDemasiadoGrandeError(Error):
    pass

numero = 10

while(True):
    try:
        numero_entrada = int(input("Introduce un numero: "))
        if(numero_entrada < numero):
            raise ValorDemasiadoPequeñoError
        elif(numero_entrada > numero):
            raise ValorDemasiadoGrandeError
        break
    except ValorDemasiadoPequeñoError:
        print("El numero es demasiado pequeño, intentalo de nuevo")
        print()
    except ValorDemasiadoGrandeError:
        print("El numero es demasiado grande, intentalo de nuevo")
        print()

print("Has encontrado el numero correcto, enhorabuena")
```

```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion1
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$ python excepcionesPersonalizadas.py
Introduce un numero: 2
El numero es demasiado pequeño, intentalo de nuevo

Introduce un numero: 8
El numero es demasiado pequeño, intentalo de nuevo

Introduce un numero: 15
El numero es demasiado grande, intentalo de nuevo

Introduce un numero: 13
El numero es demasiado grande, intentalo de nuevo

Introduce un numero: 9
El numero es demasiado pequeño, intentalo de nuevo

Introduce un numero: 11
El numero es demasiado grande, intentalo de nuevo

Introduce un numero: 10
Has encontrado el numero correcto, enhorabuena
(fp) (base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$
```

## 2. PRUEBAS Y VALIDACIÓN DE DATOS

La validación de datos es una técnica que permite asegurar que los valores de entrada con los que vaya a operar un programa o función de Python están dentro de un determinado dominio. Este tipo de técnicas se han convertido en una herramienta imprescindible de todo programador, ya que además de prevenir errores de ejecución derivados de utilizar datos incompatibles o fuera del dominio, también aumentan la interpretabilidad de nuestro código ya que si las validaciones han sido realizadas correctamente pueden devolver información de interés para que el usuario pueda actuar en consecuencia, solucionando un error de forma prematura.

Existen diversas formas para comprobar el dominio de un dato:

- Comprobando el contenido del dato: que una variable sea de un tipo en particular.
- Que el dato tenga una característica determinada.

Hemos de tener en cuenta que debemos pensar una forma de actuar o responder si se “dispara” alguna alarma que indique que algo no va bien, ya que es también nuestro objetivo mostrar información de interés al usuario que le ayude a detectar y solucionar el error. En estos casos, hacer uso de excepciones (del sistema o creadas por nosotros mismos) son una solución elegante.

Al uso concienzudo de este tipo de técnicas se le conoce como **programación defensiva**.



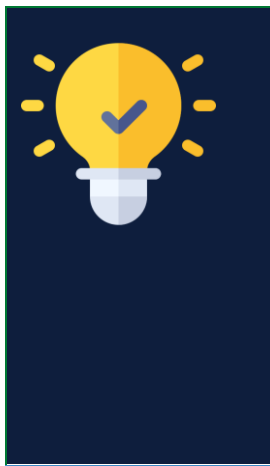
### ***Presta atención***

Una función de Python puede recibir datos de entrada por dos medios diferentes:

1. Como resultado de una función o programa de Python.
2. Valores ingresados manualmente por el usuario o por medio de un fichero de datos.

La programación defensiva persigue la mejora del software y su código fuente atendiendo a tres criterios principales:

1. **Calidad.** Adoptar medidas de programación defensiva reduce el número de fallos del software.
2. **Interpretabilidad.** Mejora la comprensibilidad y legibilidad del código, a prueba de una auditoría de código.
3. **Prevención de errores.** Permite que el software desarrollado se comporte de forma impredecible a pesar de que el usuario o terceras personas realicen acciones inesperadas sobre nuestro código.



### **Importante**

Los errores de software (bugs) pueden ser potencialmente utilizados por hackers para una inyección de código, ataques de denegación de servicio u otro ataque.

## **2.1 Técnicas de programación defensiva**

Existen diversas técnicas de programación defensiva que un desarrollador puede adoptar para mejorar la calidad de su código. A continuación se muestran las más conocidas o adoptadas por la comunidad de científicos de computación:

1. **Revisiones de código fuente.** Una revisión de código se refiere a que alguien diferente al autor original del código realice una auditoría del mismo. Es conveniente que la auditoría sea realizada por alguien ajeno al desarrollo del código. Hacer el código disponible (usando plataformas como github por ejemplo) es una buena medida, aunque no es suficiente ya que no podemos tener la certeza de que el código sea debidamente revisado.



2. **Pruebas de software.** Crear test unitarios para comprobar que un determinado programa cumple su función y devuelve valores apropiados para una determinada entrada. Lo estudiaremos en detalle en la lección 2.
3. **Reducir la complejidad del código fuente.** Evitar hacer implementaciones complejas. Realiza el código de la manera más sencilla posible para que resuelva su función.
4. **Reutilización del código fuente.** Siempre que sea posible, reutiliza el código. De nuevo, la idea es evitar crear nuevos bugs.
5. **Problemas de legado.** Antes de reutilizar código fuente antiguo, bibliotecas o APIs, éste debe ser analizado y estudiado en detalle, considerando si las funcionalidades externas que queremos incluir en nuestro código son aptas para ser reutilizadas o si son propensas a problemas de legado.



### ***Presta atención***

Implementaciones de software muy complejas aumentan la aparición de bugs. Los bugs están directamente relacionados con problemas de seguridad.



### ***Presta atención***

Los problemas de legado son problemas inherentes cuando se espera que viejos diseños trabajen con los requerimientos actuales, sobre todo cuando estos diseños no fueron diseñados con estos requerimientos en mente.

## 2.2 Validaciones: caso práctico

En esta sección mostraremos un conjunto de casos en los que resulta de interés desarrollar un conjunto de comprobaciones y validaciones. No obstante, existe una multitud de casos en los que pueden desarrollarse sistemas de validación de código. Es por ello que el objetivo de esta lección es el de mostrar al estudiante en qué consiste las pruebas y validación de datos, cuáles son sus beneficios y algunos ejemplos prácticos. Es misión del desarrollador (estudiante) determinar cuándo es necesario incluir validaciones en su código y actuar en consecuencia.

## 2.3 Comprobaciones por contenido

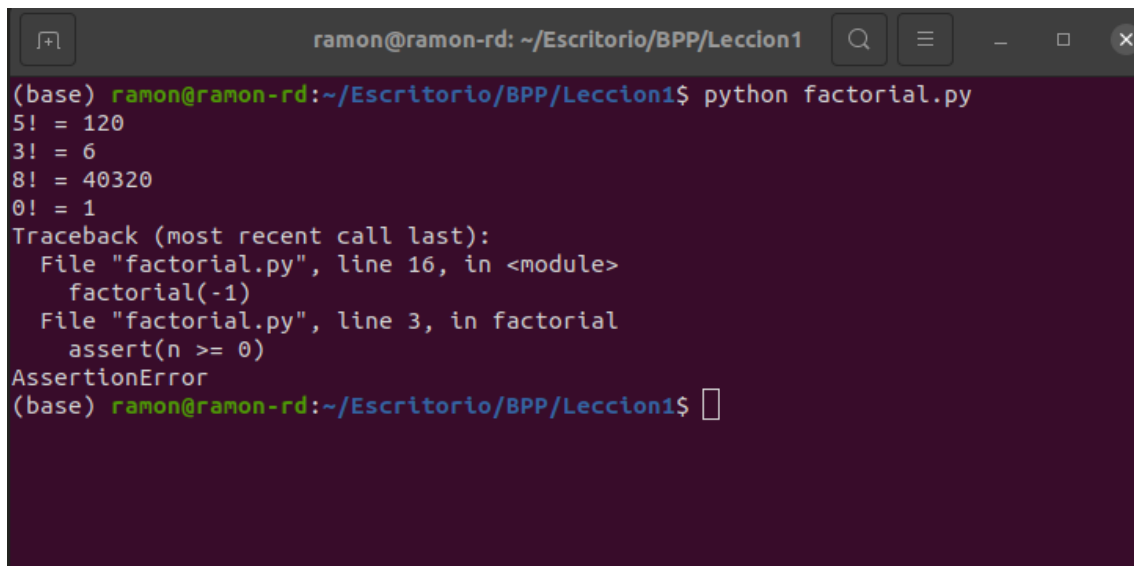
Estas comprobaciones se utilizan cuando queremos validar que los datos recibidos como entrada para realizar una determinada operación contienen información apropiada, ya sea porque el usuario ingresara información errónea o porque por cualquier motivo la información de entrada sea incorrecta. Por ejemplo, en el caso en que queramos realizar una función que realice la división de dos números enteros debemos comprobar que el denominador es distinto de cero.

Estas comprobaciones no podremos realizarlas siempre, ya que en algunas ocasiones puede ser muy laborioso comprobar si la entrada es correcta. Es por ello que el desarrollador debe prestar atención a sus implementaciones y determinar cuándo utilizar este tipo de comprobaciones.

En el siguiente ejemplo, se muestra un fragmento de código que calcula el factorial de un número. En este código utilizamos el comando *assert* para comprobar que el contenido de la variable de entrada es mayor o igual que 0.

```
def factorial(n):  
    assert(n >= 0)  
  
    fct = 1  
    for i in range(1, n+1):  
        fct *= i  
  
    print(f"{n}! = {fct}")
```

Algunos ejemplos de ejecución, incluyendo la inserción de valores no permitidos:



```
(base) ramon@ramon-rd: ~/Escritorio/BPP/Leccion1$ python factorial.py
5! = 120
3! = 6
8! = 40320
0! = 1
Traceback (most recent call last):
  File "factorial.py", line 16, in <module>
    factorial(-1)
  File "factorial.py", line 3, in factorial
    assert(n >= 0)
AssertionError
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$
```

## 2.4 Comprobación de valores de entrada del usuario

Imaginad que necesitamos desarrollar una función que solicite al usuario que introduzca por teclado un número entero. El usuario, de forma involuntaria o maliciosa, podría insertar un valor diferente a un entero, por ejemplo una cadena de caracteres.

Para manejar este tipo de errores, podemos hacer uso de la función *input* combinado con el manejo de excepciones. Esta función almacena el dato (o datos) introducido por el usuario como una cadena y posteriormente podemos transformarla a un entero. Si el valor que introduce el usuario no es un entero, le damos alguna oportunidad extra para que ingrese un valor correcto. Si tras 3 intentos no ingresa un valor adecuado, suponemos que el usuario no tiene buenas intenciones y terminamos el programa.

```
def leer_enteros():
    num_intentos = 0
    while(num_intentos < 3):
        n = input("Introduzca un número entero: ")
        try:
            n = int(n)
            #Aquí podemos ingresar cualquier fragmento de código que opere con n
            return n
        except ValueError:
            num_intentos += 1
    raise(ValueError, "Has ingresado un valor incorrecto en tres ocasiones. Se acabaron tus oportunidades.")
```

```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion1
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$ python lee_enteros.py
Introduzca un número entero: a
Introduzca un número entero: 2.25
Introduzca un número entero: a6
Traceback (most recent call last):
  File "lee_enteros.py", line 17, in <module>
    leer_enteros()
  File "lee_enteros.py", line 13, in leer_enteros
    raise(ValueError, "Has ingresado un valor incorrecto en tres ocasiones. Se a
cabaron tus oportunidades.")
TypeError: exceptions must derive from BaseException
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$
```

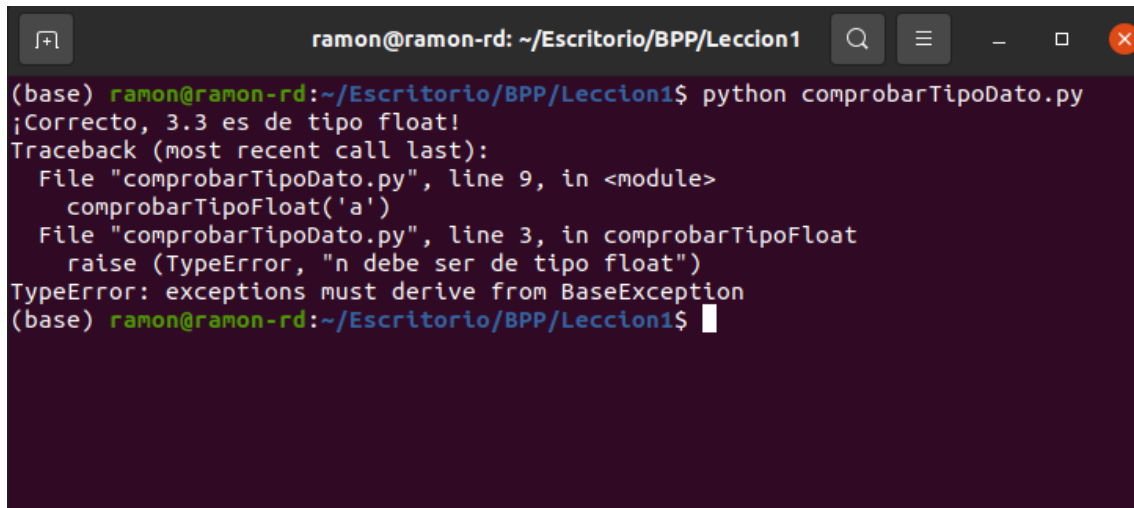
## 2.5 Comprobaciones por tipo

Otra forma de realizar las comprobaciones por tipo es mediante el uso de la función `type`. Esta función recibe por parámetro un valor o una variable y devuelve su tipo.

Podemos comprobar que el tipo de una variable es de tipo flotante:

```
def comprobarTipoFloat(n):
    if(type(n) != float):
        raise (TypeError, "n debe ser de tipo float")
    else:
        print(f"¡Correcto, {n} es de tipo float!")
```

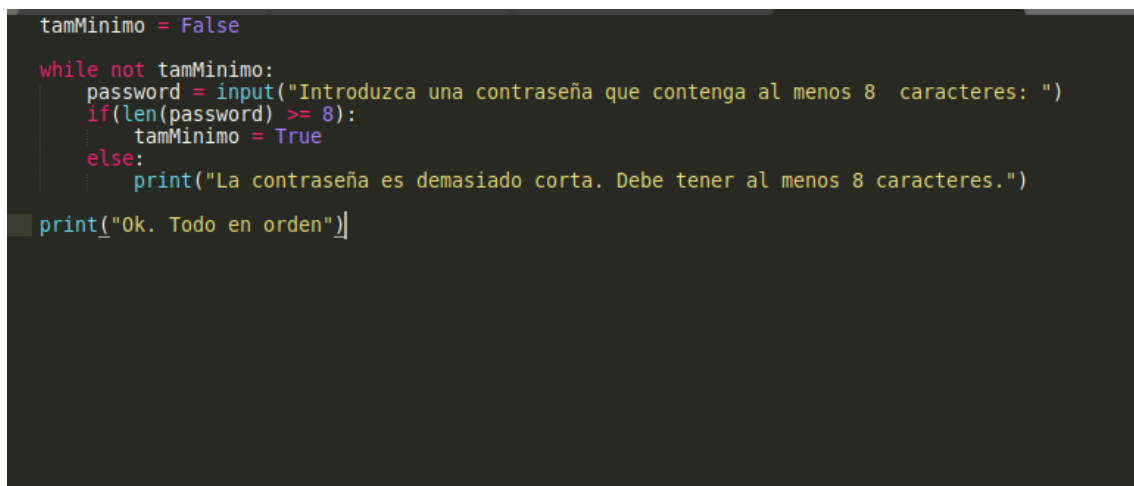
Algunos ejemplos de ejecución:



```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion1
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$ python comprobarTipoDato.py
¡Correcto, 3.3 es de tipo float!
Traceback (most recent call last):
  File "comprobarTipoDato.py", line 9, in <module>
    comprobarTipoFloat('a')
  File "comprobarTipoDato.py", line 3, in comprobarTipoFloat
    raise (TypeError, "n debe ser de tipo float")
TypeError: exceptions must derive from BaseException
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$
```

## 2.6 Comprobaciones por tamaño de cadena

En algunas aplicaciones resulta de interés incluir un mecanismo de validación para comprobar que una contraseña es correcta. En este ejemplo práctico (y sencillo), desarrollamos un método de validación para comprobar que la contraseña ingresada por el usuario tiene un tamaño mínimo.



```
tamMinimo = False
while not tamMinimo:
    password = input("Introduzca una contraseña que contenga al menos 8 caracteres: ")
    if(len(password) >= 8):
        tamMinimo = True
    else:
        print("La contraseña es demasiado corta. Debe tener al menos 8 caracteres.")
print("Ok. Todo en orden")]
```

```
ramon@ramon-rd: ~/Escritorio/BPP/Leccion1
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$ python password.py
Introduzca una contraseña que contenga al menos 8 caracteres: 122
La contraseña es demasiado corta. Debe tener al menos 8 caracteres.
Introduzca una contraseña que contenga al menos 8 caracteres: asdf
La contraseña es demasiado corta. Debe tener al menos 8 caracteres.
Introduzca una contraseña que contenga al menos 8 caracteres: my_pass
La contraseña es demasiado corta. Debe tener al menos 8 caracteres.
Introduzca una contraseña que contenga al menos 8 caracteres: pass_33
La contraseña es demasiado corta. Debe tener al menos 8 caracteres.
Introduzca una contraseña que contenga al menos 8 caracteres: mY_p4sS_33
Ok. Todo en orden
(base) ramon@ramon-rd:~/Escritorio/BPP/Leccion1$
```

### 3. PUNTOS CLAVE

En esta lección hemos aprendido:

- | Las herramientas disponibles para llevar a cabo el control de errores en nuestro código.
- | Cómo gestionar y depurar errores de un código de programación por medio del manejo de excepciones.
- | Cómo llevar a cabo el concepto de programación defensiva a nuestras implementaciones, creando software robusto y de mayor calidad.

